**Andrew Finn**

**184XXXXX**

Before writing any prolog i first went and found a way to solve the problem in Python. I found an implementation of Dijkstra’s algorithm for Python[1] as well as Prolog[2 & 3] I then used the python version to first solve the question in python first, the source code for Python implementation can be seen [here](https://github.com/Andrew-Finn/Daily-Coding-Problems/blob/master/Random/dijkstra.py)[4]. I also used a YouTube [video from NumberPhile](https://www.youtube.com/watch?v=GazC3A4OQTE) to understand how Dijkstra’s algorithm functions[5]..

I first made a rule to define the various transport method available and their average speed.

| speed(f,8). speed(c,100). speed(t,120). speed(p,750).  *Predefined*  route(dublin, cork, 200, 'fct'). route(cork, dublin, 200, 'fct'). route(cork, corkAirport, 20, 'fc'). route(corkAirport, cork, 25, 'fc'). route(dublin, dublinAirport, 10, 'fc'). route(dublinAirport, dublin, 20, 'fc'). route(dublinAirport, corkAirport, 225, 'p'). route(corkAirport, dublinAirport, 225, 'p').  Out\_nodes generates all reachable nodes R from X using given methods  out\_nodes(Modes,X,R):-  out\_nodes\_rec(Modes,X,[],R).  Atom\_chars converts the transport methods from atoms to a list of characters.[6]  out\_nodes\_rec(Modes,X,A,Res):-  route(X,N,\_,RouteModes),  atom\_chars(Modes,ModeList),  atom\_chars(RouteModes,RouteModeList),  intersection(ModeList,RouteModeList,[\_|\_]),  \+ member(N,A),!,  out\_nodes\_rec(Modes,X,[N|A],Res). out\_nodes\_rec(\_,\_,A,A).  The time and fastest\_mode functions are used to find the time needed to go from A to B with given methods in terms of time this is needed as we can then use Time as the cost parameter in Dijkstra’s algorithm. time(Modes,From,To,Time):-  atom\_chars(Modes,ModeList),  route(From,To,Distance,RouteModes),  atom\_chars(RouteModes,RouteModeList),  intersection(ModeList,RouteModeList,CommonModes),  fastest\_mode\_list(CommonModes,MaxMode),  speed(MaxMode,Speed),  Time is Distance/Speed.  fastest\_mode\_list([Mode|Modes],MaxMode):-  fastest\_mode\_rec(Modes,Mode,MaxMode). fastest\_mode\_rec([Mode|Modes],TmpMax0,MaxMode):-  fastest\_mode(Mode,TmpMax0,TmpMax1),  fastest\_mode\_rec(Modes,TmpMax1,MaxMode). fastest\_mode\_rec([],MaxMode,MaxMode). fastest\_mode(Mode1,Mode2,Mode1):-  speed(Mode1,Speed1),  speed(Mode2,Speed2),  Speed1 > Speed2, !. fastest\_mode(\_,Mode,Mode).  Dijkstra’s algorithm uses a Priority Queue to store its progress and paths already considered as well as to prioritize paths with shorter costs (Not foolproof). Below is the implementation of this queue as well as functions needed to modify the queue. Elements of queues are triplets.  queue\_node\_time(Node,[[Node,\_,Time]|\_],Time):- !. queue\_node\_time(Node,[\_|T],Time):-  queue\_node\_time(Node,T,Time). queue\_node\_time(\_,[],0).  queue\_node\_parent(\_,[],nil). queue\_node\_parent(Node,[[Node,Parent,\_]|\_],Parent):- !. queue\_node\_parent(Node,[\_|T],Parent):-  queue\_node\_parent(Node,T,Parent).  queue\_has\_node([[Node,\_,\_]|\_],Node):- !. queue\_has\_node([\_|T],Node):-  queue\_has\_node(T,Node).  queue\_insert\_node(Node,Parent,Time,[Head|Tail],[Head|Tail1]):-  Head = [\_,\_,Time1],  Time > Time1, !,  queue\_insert\_node(Node,Parent,Time,Tail,Tail1). queue\_insert\_node(Node,Parent,Time,[Head|Tail],[[Node,Parent,Time],Head|Tail]):-  Head = [\_,\_,Time1],  Time =< Time1, !. queue\_insert\_node(Node,Parent,Time,[],[[Node,Parent,Time]]).  queue\_delete\_node(Node,[[Node,\_,\_]|Tail],Tail):- !. queue\_delete\_node(Node,[Head0|Tail0],[Head0|Tail1]):-  queue\_delete\_node(Node,Tail0,Tail1).  Dijkstra’s algorithm uses a queue of nodes sorted by cost to the start node. Every time we process a node we remove it from the queue and add it to ListDone. ListDone contains a list of all nodes previously visited and as a result know the shortest path to. We can then repeat this starting for the first node in the queue. The outer loop of the algorithm takes the first node from the queue and process it, the inner loop takes neighbours of the previous node and add them to the queue or updates their position respective of cost if already in the queue. This is repeated until the all nodes are processed.  node(Node,Goal,Modes,Queue,ListDone,Res):-  Node = [NodeName,\_,\_],  NodeName \= Goal,  out\_nodes(Modes,NodeName,Neighbors),  \+ queue\_has\_node(ListDone,NodeName),  update\_queue(Node,Modes,Neighbors,ListDone,Queue,Queue1), !,  Queue1 = [Next|Queue2],  node(Next,Goal,Modes,Queue2,[Node|ListDone],Res). node(Node,Goal,\_,\_,ListDone,Res):-  Node = [NodeName,\_,\_],  NodeName = Goal,  reverse([Node|ListDone],Res).  update\_queue(\_,\_,[],\_,Queue,Queue). update\_queue(ParentNode,Modes,[Node|Nodes],ListDone,QueueIn,QueueOut):-  queue\_has\_node(ListDone,Node), !,  update\_queue(ParentNode,Modes,Nodes,ListDone,QueueIn,QueueOut). update\_queue(ParentNode,Modes,[Node|Nodes],ListDone,QueueIn,QueueOut):-  ParentNode = [ParentName,\_,ParentTime],  time(Modes,ParentName,Node,NodeParentRouteTime),  NodeTime is ParentTime+NodeParentRouteTime,  update\_queue\_node(Modes,ParentName,Node,NodeTime,QueueIn,Queue1),  update\_queue(ParentNode,Modes,Nodes,ListDone,Queue1,QueueOut).  update\_queue\_node(\_,Parent,Node,NodeTime,QueueIn,QueueOut):-  queue\_has\_node(QueueIn,Node), !,  queue\_node\_time(Node,QueueIn,OldTime),  ( NodeTime < OldTime  ->  queue\_delete\_node(Node,QueueIn,Queue1),  queue\_insert\_node(Node,Parent,NodeTime,Queue1,QueueOut)  ; QueueOut = QueueIn ). update\_queue\_node(\_,Parent,Node,NodeTime,QueueIn,QueueOut):-  queue\_insert\_node(Node,Parent,NodeTime,QueueIn,QueueOut).  Gen\_path reconstructs the list of the nodes from ListDone to give us our shortest path.  gen\_path(From,To,Nodes,Path):-  make\_rpath(From,To,Nodes,RPath),  reverse(RPath,Path). make\_rpath(N,N,\_,[N]):- !. make\_rpath(From,To,Nodes,[To|Path]):-  queue\_node\_parent(To,Nodes,Prev),  make\_rpath(From,Prev,Nodes,Path).  journey(S,D,M):-  node([S,nil,0],D,M,[],[],R),  gen\_path(S,D,R,Path),  writeln(Path). |
| --- |

Dijkstra will always find the shortest path however has it no herestics involved so it can be throw down a very long and costly long path. Howver for the purpose of the sampe input Dijkstra is suffecient.

References:

[1] - https://dev.to/mxl/dijkstras-algorithm-in-python-algorithms-for-beginners-dkc

[2] - http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm

[3] - https://github.com/wvrossem/Prolog-Dijkstra-Algorithm/blob/master/src/dijkstra.pro

[4] - https://github.com/**Andrew-Finn**/Daily-Coding-Problems/blob/master/Random/dijkstra.py

[5] - <https://www.youtube.com/watch?v=GazC3A4OQTE>

[6] - https://stackoverflow.com/questions/7647758/prolog-findall-implementation